
# S-RASTER: contraction clustering for evolving data streams

Journal of Big Data

RESEARCH Open Access

# S-RASTER: contraction clustering for evolving data streams

Gregor Ulm[1,2*], Simon Smith[1,2], Adrian Nilsson[1,2], Emil Gustavsson[1,2] and Mats Jirstrand[1,2]

*Correspondence:
gregor.ulm@fcc.chalmers.se
[1] Fraunhofer-Chalmers Research Centre for Industrial Mathematics, Chalmers Science Park, 412 88 Gothenburg, Sweden
Full list of author information is available at the end of the article

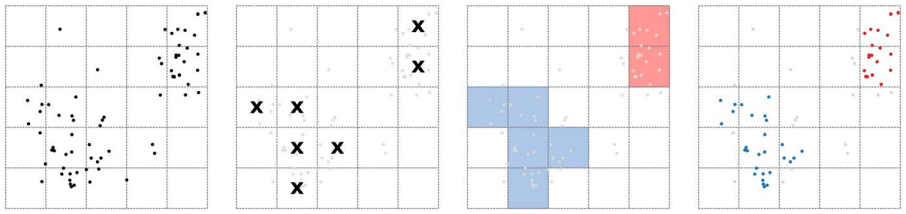
**Abstract**

Contraction Clustering (RASTER) is a single-pass algorithm for density-based clustering of 2D data. It can process arbitrary amounts of data in linear time and in constant memory, quickly identifying approximate clusters. It also exhibits good scalability in the presence of multiple CPU cores. RASTER exhibits very competitive performance compared to standard clustering algorithms, but at the cost of decreased precision. Yet, RASTER is limited to batch processing and unable to identify clusters that only exist temporarily. In contrast, S-RASTER is an adaptation of RASTER to the stream processing paradigm that is able to identify clusters in evolving data streams. This algorithm retains the main benefits of its parent algorithm, i.e. single-pass linear time cost and constant memory requirements for each discrete time step within a sliding window. The sliding window is efficiently pruned, and clustering is still performed in linear time. Like RASTER, S-RASTER trades off an often negligible amount of precision for speed. Our evaluation shows that competing algorithms are at least 50% slower. Furthermore, S-RASTER shows good qualitative results, based on standard metrics. It is very well suited to real-world scenarios where clustering does not happen continually but only periodically.

**Keywords:** Big data, Stream processing, Clustering, Machine learning, Unsupervised learning, Big data analytics

## Introduction

Clustering is a standard method for data analysis and many clustering methods have been proposed [29]. Some of the most well-known clustering algorithms are DBSCAN [9], *k*-means clustering [23], and CLIQUE [1, 2]. Yet, they have in common that they do not perform well with big data, i.e. data that far exceeds available main memory [34]. This was also confirmed by our own experience when we faced the real-world industrial challenge of identifying dense clusters in terabytes of geospatial data. This led us to develop Contraction Clustering (RASTER), a very fast linear-time clustering algorithm for identifying approximate density-based clusters in 2D data, primarily motivated by the fact that existing batch processing algorithms for this purpose exhibited insufficient performance. We previously described RASTER and highlighted its performance for sequential processing of batch data [32]. This was followed by a description of a parallel version of that algorithm [33]. A key aspect of RASTER (cf. Fig. 1) is that it does not



(a) Input data (b) Significant tiles (c) Clustered tiles (d) Clustered points

**Fig. 1** High-level visualization of RASTER (best viewed in color), using a simplified example on a small 5 × 5 grid. Please also refer the section "RASTER" and Table 1. The precision of the input is reduced, with leads to an implied grid. This grid is shown to aid the reader but it is not explicitly constructed by the algorithm. The original input is shown in **a**, followed by projection to tiles in **b** where only significant tiles that contain at least $\tau = 4$ values are retained. Tiles that contain less than $\tau$ values are subsequently ignored as they are treated as noise. The result is a set of significant tiles. The parameter $\mu$ specifies how many significant tiles a cluster has to contain as a minimum. In this case, given a minimum cluster size of $\mu = 2$ and a maximum distance $\delta = 1$, i.e. significant tiles need to be adjacent, two clusters emerge (cf. **c**), which corresponds to RASTER. Clusters as collections of points are shown in **d**, which corresponds to the variant RASTER′

exhaustively cluster its input but instead identifies their approximate location in linear time. As it only requires constant space, it is eminently suitable for clustering big data. The variant RASTER′ retains its input and still runs in linear time while requiring only a single pass. Of course, it cannot operate in constant memory.

A common motivation for stream processing is that data does not fit into working memory and therefore cannot be retained. This is not a concern for RASTER as it can process an arbitrary amount of data in limited working memory. One could therefore divide a stream of data into discrete batches and consecutively cluster them. Yet, this approach does not address the problem that, in a given stream of data, any density-based cluster may only exist temporarily. In order to solve this problem, this paper presents Contraction Clustering for Evolving Data Streams (S-RASTER). This algorithm has been designed for identifying density-based clusters in infinite data streams within a sliding window. S-RASTER is not a replacement of RASTER, but a complement, enabling this pair of algorithms to efficiently cluster data, regardless of whether it is available as a batch or a stream.

Given that there is already a number of established algorithms for detecting clusters in data streams, the work on S-RASTER may need to be further motivated. The original motivation is related to the batch processing clustering algorithm RASTER, which is faster than competing algorithms and also requires less memory. This comes at the cost of reduced precision, however. For the use case we have been working on, i.e. very large data sets at the terabyte level, even modest improvements in speed or memory requirements compared to the status quo were a worthwhile pursuit. Indeed, RASTER satisfied our use case in industry as it is faster than competing algorithms and also requires less memory. Yet, we wanted to improve on it as it did not help us to process data streams. It also is not able to detect clusters that only exist temporarily. Furthermore, we also wanted to explore if there is a faster way to process the data we needed to process, given our constraints (cf. Sect. "Identifying evolving hubs"). These reasons motivated our work on S-RASTER. The goal was to achieve faster data processing than existing methods allow, using only a single pass, but with an acceptable loss of precision. As the results in this paper show, S-RASTER is, in a standard benchmark, indeed faster than competing algorithms for clustering data streams.

In the remainder of this paper, we provide relevant background in the Sect. "Background", which contains a brief recapitulation of RASTER and the motivating use case for S-RASTER, i.e. identifying evolving hubs in streams of GPS data. In the Sect. "S-RASTER" we provide a detailed description of S-RASTER, including a complete specification in pseudocode. This is followed by a theoretical evaluation of S-RASTER in the Sect. "Theoretical evaluation" and a description of our experiments in the Sect. "Experiment". The results of our experiments as well as a discussion of them are presented in the Sect. "Results and discussion". Some related work is part of the evaluation section as we compare S-RASTER to competing algorithms. However, further related work is highlighted in the Sect. "Related work" and future work in "Future work" section. We finish with a conclusion in "Conclusions" section.

## Background

In this section, we give a brief presentation of the sequential RASTER algorithm in the Sect. "RASTER". This is followed by a description of the motivating problem behind S-RASTER, i.e. the identification of so-called hubs within a sliding window, in the Sect. "Identifying evolving hubs".

### RASTER

In this subsection, we provide a brief description of RASTER [32, 33]. This algorithm approximately identifies density-based clusters very quickly (cf. Alg. 1). The main idea is to project data points to tiles and keep track of the number of points that are projected to each tile. Only tiles to which more than a predefined threshold number $\tau$ of data points have been projected are retained. These are referred to as significant tiles $\sigma$, which are subsequently clustered by exhaustive lookup of neighboring tiles in a depth-first manner. Clustering continues for as long as there are significant tiles left. To do so, the algorithm selects an arbitrary tile as the seed of a new cluster. This cluster is grown iteratively by looking up all neighboring tiles within a given Manhattan or Chebyshev distance $\delta$. This takes only $\mathcal{O}(1)$ as the location of all potential neighbors is known due to their location in the grid. Only clusters that contain more than a predefined number $\mu$ of significant tiles are retained. While RASTER was developed to identify dense clusters, it can identify clusters that have irregular shapes as well. Refer to Fig. 2 for an illustration.

The projection operation consists of reducing the precision of the input by scaling a floating-point number to an integer. For instance, take an arbitrary GPS coordinate (34.59204302, 106.36527351), which is already truncated compared to the full representation with double-precision floating-point numbers. GPS data is inherently imprecise, yet stored in floating-point format with the maximum precision, which is potentially misleading, considering that consumer-grade GPS is only accurate to within about 5 to 10 m in good conditions, primarily open landscapes [8, 36]. In contrast, in urban landscapes GPS accuracy tends to be much poorer. A study based on London, for instance, found a mean error of raw GPS measurements of vehicular movements of 53.7 m [30]. Consequently, GPS data suggests a level of precision they do not possess. Thus, by dropping a few digit values, we do not lose much, if any, information. Furthermore, vehicle GPS data is sent by vehicles that may, in the case of trucks with an attached trailer, be more than 25 m long. To cluster such coordinates, we can truncate even more digit points. For instance, if four place values after the decimal point are enough, which corresponds to a resolution

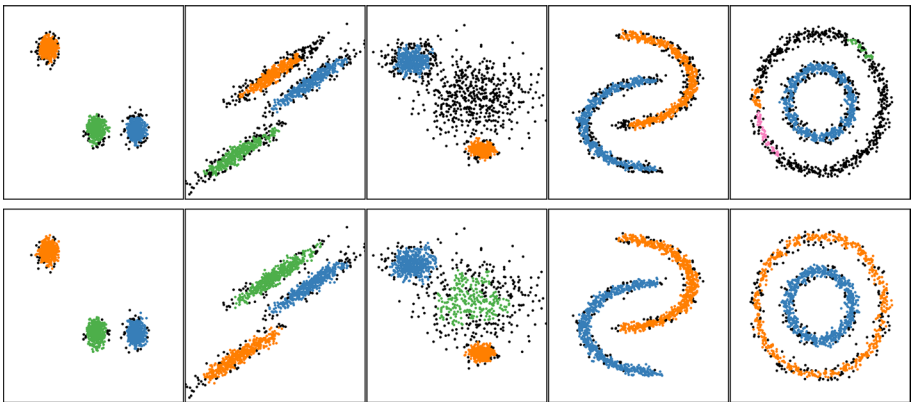


**Fig. 2** The precision parameter $\xi$ greatly influences clustering results of RASTER (best viewed in color). This illustration is based on retaining all data points (cf. Fig. 1d). With a precision of $\xi = 0.90$ (top), all but the rightmost data set are clustered satisfactorily. Reducing the precision to $\xi = 0.73$ (bottom) improves the results of that data set. It is a matter of debate which value of $\xi$ led to a better result with the data set in the middle as a good case could be made for either result, depending on whether the goal of the user is to identify dense or sparse clusters. The data sets were taken from a collection of standard data sets for the evaluation of general-purpose clustering algorithms that are part of the machine learning library `scikit-learn` [27]. A more extensive discussion of these results is provided in a previous paper on RASTER [33]

of 11.1 m in the case of GPS, we transform the aforementioned sample data point to (34.5920, 106.3652). However, to avoid issues pertaining to working with floating-point numbers, the input is instead scaled to (345920, 1063652). Only before generating the final output, all significant tiles of the resulting clusters are scaled back to floating-point numbers, i.e. the closest floating-point representation of (34.592, 106.3652).

**Algorithm 1** RASTER

**input:** data *points*, precision $\xi$, threshold $\tau$, distance $\delta$, minimum cluster size $\mu$
**output:** set of clusters *clusters*

1: $acc := \varnothing$ ▷ $\{(x_\pi, y_\pi) : count\}$
2: $clusters := \varnothing$ ▷ set of sets
3: **for** $(x, y)$ in *points* **do**
4: $(x_\pi, y_\pi) := project(x, y, \xi)$ ▷ $\mathcal{O}(1)$
5: **if** $(x_\pi, y_\pi) \notin$ keys of $acc$ **then**
6: $acc[(x_\pi, y_\pi)] := 1$
7: **else**
8: $acc[(x_\pi, y_\pi)]$ += 1
9: **for** $(x_\pi, y_\pi)$ in $acc$ **do**
10: **if** $acc[(x_\pi, y_\pi)] < \tau$ **then**
11: remove $acc[(x_\pi, y_\pi)]$
12: $\sigma :=$ keys of $acc$ ▷ significant tiles
13: **while** $\sigma \neq \varnothing$ **do** ▷ $\mathcal{O}(n)$ for lls. 12–24
14: $t := \sigma.pop()$
15: $cluster := \varnothing$ ▷ set
16: $visit := \{t\}$
17: **while** $visit \neq \varnothing$ **do**
18: $u := visit.pop()$
19: $ns := neighbors(u, \delta)$ ▷ $\mathcal{O}(1)$
20: $cluster := cluster \cup \{u\}$
21: $\sigma := \sigma \setminus ns$ ▷ cf. ln. 13
22: $visit := visit \cup ns$
23: **if** size of $cluster \geq \mu$ **then**
24: add *cluster* to *clusters*

Two notes regarding the reduction of precision are in order. First, this procedure is not limited to merely dropping digit values. As those values are only intermediary representations that are used for clustering, any non-zero real number can be used as the scaling factor. Second, the magnitude of the scaling factor depends on two aspects, precision of the provided data and size of the objects we want to identify clusters of. For the former, it should be immediately obvious that data that suggests a greater precision than it actually possesses, like GPS data mentioned above, can be preprocessed accordingly without any loss of information. The latter depends on the domain and the trade-off the user is willing to make as both clustering speed and memory requirements directly depend on the chosen precision.

As clustering belongs to the field of unsupervised learning, there are potentially multiple satisfactory clusterings possible with any given dataset. With RASTER, the user can influence clustering results by adjusting four parameters (cf. Table 1): precision $\xi$, threshold for a significant tile $\tau$, maximum distance of significant tiles in a cluster $\delta$, and threshold for the cluster size $\mu$. The precision parameter $\xi$ directly influences the granularity of the implied grid (cf. Fig. 1). A lower precision value for $\xi$ leads to a coarser grid, and vice versa. With the value $\tau$, it is possible to directly influence the number of significant tiles that are detected in any given data set. The higher this value, the fewer significant tiles will be identified. While the most intuitive value for the distance parameter $\delta$ is 1, meaning that all significant tiles that are combined when constructing a cluster need to be direct neighbors, it is possible to also take the case of sparser clusters into account. For instance, if the user detects two dense clusters that are only one tile apart, it may make sense to set $\delta = 2$ to combine them. Of course, this depends on the application domain. Lastly, the parameter $\mu$ determines when a collection of significant tiles is considered a cluster. The higher this value, the fewer clusters will be detected.

RASTER is a single-pass linear time algorithm. However, in a big data context, its biggest benefit is that it only requires constant memory, assuming a finite range of inputs. This is the case with GPS data. It is therefore possible to process an arbitrary amount of data on a resource-constrained workstation with this algorithm. We have also shown that it can be effectively parallelized [33]. A variation of this algorithm that retains its inputs is referred to as RASTER′. It is less suited for big data applications. However, it is effective for general-purpose density-based clustering and very competitive compared to standard clustering methods; cf. Appendix A in [33].

#### Identifying evolving hubs

RASTER was designed for finite batches of GPS traces of commercial vehicles. The goal was to identify hubs, i.e. locations where many vehicles come to a halt, for instance vans at delivery points or buses at bus stops. After identifying all hubs in a data set, it is possible to construct vehicular networks. However, what if the data does not represent a static reality? It is a common observation that the location of hubs changes over time. A bus stop may get moved or abolished, for instance. This motivates modifying RASTER so that it is able to detect hubs over time and maintaining hubs within a sliding window $W$. The length of $W$ depends on the actual use case. With GPS traces of infrequent but important deliveries, many months may be necessary. Yet, with daily deliveries, a few days would suffice to detect new hubs as well as discard old ones.

**Table 1 Overview of symbols used in the description of RASTER and S-RASTER**

| Symbol | Meaning |
|---|---|
| $\xi$ | Precision for projection operation |
| $\tau$ | Threshold number of points to determine if a tile is significant |
| $\delta$ | Distance metric for cluster definition |
| $\mu$ | Minimum cluster size in terms of the number of significant tiles |
| $\pi$ | Projection operator |
| $\alpha$ | Accumulation operator |
| $\kappa$ | Clustering operator |

The parameters $\xi$, $\tau$, $\delta$, and $\mu$ are used for both RASTER and S-RASTER and directly influence clustering results. In contrast, the symbols $\pi$, $\alpha$, and $\kappa$ are only used for the description of S-RASTER

## S-RASTER

This section starts with a concise general description of S-RASTER in the Sect. "Idea", followed by a detailed specification in the Sect. "Detailed description". Afterwards, we highlight some implementation details in the Sect. "Implementation details" and outline, in the Sect. "Retaining data points with S-RASTER′", how S-RASTER has to be modified to retain its inputs, which makes this algorithm applicable to different use cases.

### Idea

Before we present a more technical description of S-RASTER, we would like to start with a more intuitive description of this algorithm, using GPS coordinates for the purpose of illustration. For the sake of simplicity, we also ignore parallel computations for the time being. Imagine five sequential nodes: source ($s$), projection ($\pi$), accumulation ($\alpha$), clustering ($\kappa$), and sink ($t$). The general idea is that S-RASTER processes input data using those five nodes. Input is provided by the source $s$ and, without modification, forwarded to node $\pi$, which performs projections, e.g. it reduces the precision of the provided input by scaling it. A simple example consists of dropping place values of GPS coordinates (cf. Sect. "RASTER"). The projected values are sent from node $\pi$ to node $\alpha$, which keeps track of the number of points that were projected to each tile in the input space for the duration of the chosen window. Once the status of a tile changes, i.e. it becomes significant or was once significant but no longer is, an update is sent to node $\kappa$. These steps happen continually whenever there is new data to process. In contrast, node $\kappa$ performs clustering based on significant tiles in a fixed interval, which is followed by sending clusters as sets of significant tiles to the sink node $t$.

More formally, S-RASTER (cf. Fig. 3) performs, for an indefinite amount of time, projection and accumulation continually, and clustering periodically. Projection nodes $\boldsymbol{\pi}$ receive their input from the source node $s$. Each incoming pair $(x, y)$, where $x, y \in \mathbb{R}$, is surjected to $(x_\pi, y_\pi)$, where $x_\pi, y_\pi \in \mathbb{Z}$. Together with a period indicator $\Delta_z$, these values are sent to accumulation nodes $\boldsymbol{\alpha}$. The identifier $\Delta_z \in \mathbb{N}_0$ designates

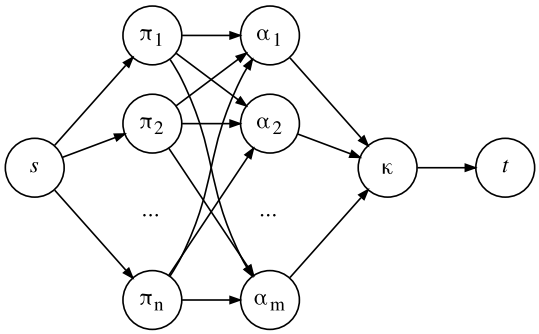



**Fig. 3** Flow graph of S-RASTER, a modification of RASTER (cf. Fig. 1) for evolving data streams. Refer to the Sect. "Idea" for a detailed description. The input source node $s$ distributes values arbitrarily to projection nodes $\boldsymbol{\pi}$, which reduce the precision of the input. In turn, they send projected values to accumulation nodes $\boldsymbol{\alpha}$. These nodes keep a count of points for each tile and determine significant tiles for the chosen sliding window. Should a tile become significant or a once significant tile no longer be significant, a corresponding update is sent to the clustering node $\kappa$. Node $\kappa$ periodically performs clustering of significant tiles, which is a very fast operation

a period with a fixed size, e.g. one day, and is non-strictly increasing. Each $\alpha$-node maintains a sliding window $W$ of length $c$, which is constructed from $c$ multisets $W_{\Delta_z}$. Each such multiset $W_{\Delta_z}$ keeps running totals of how many times the input was surjected to any given tuple $(x_\pi, y_\pi)$ in the chosen period. The sliding window starting at period $\Delta_i$ is defined as $W_{\Delta_i}^{\Delta_{i+c}} = \bigcup_{z=i}^{i+c} W_{\Delta_z}$.[1] It contains the set of significant tiles $\sigma = \{(x_\pi, y_\pi)^d \in W_{\Delta_i}^{\Delta_{i+c}} \mid d \geq \tau\}$, where $d$ indicates the multiplicity, i.e. the number of appearances in the multiset, and $\tau$ the threshold for a significant tile. If a tile becomes significant, its corresponding value $(x_\pi, y_\pi)$ is forwarded to the clustering node $\kappa$. Whenever $W$ advances from $W_{\Delta_i}^{\Delta_{i+c}}$ to $W_{\Delta_{i+1}}^{\Delta_{i+c+1}}$, the oldest entry $W_{\Delta_i}$ is removed from $W$. Furthermore, all affected running totals are adjusted, which may lead to some significant tiles no longer being significant. If so, node $\kappa$ receives corresponding updates to likewise remove those entries. Node $\kappa$ keeps track of all significant tiles, which it clusters whenever $W$ advances (cf. Alg. 1, lls. 12–24). The preliminary set of clusters is $ks$. The final set of clusters is defined as $\{k \in ks \mid k > \mu\}$, where $\mu$ is the minimum cluster size. Each $(x_\pi, y_\pi)$ in each $k$ is finally projected to $(x'_\pi, y'_\pi)$, where $x'_\pi, y'_\pi \in \mathbb{R}$. Together with cluster and period IDs, these values are sent to the sink node $t$.

### Detailed description

S-RASTER processes the data stream coming from source $s$ in Fig. 3 and outputs clusters to sink $t$. There are three different kinds of nodes: projection nodes $\boldsymbol{\pi}$ project points to tiles, accumulation nodes $\boldsymbol{\alpha}$ determine significant tiles within each sliding window $W_{\Delta_i}^{\Delta_{i+c}}$, and one clustering node $\kappa$ outputs, for each $W_{\Delta_i}^{\Delta i+c}$, all identified clusters. Below, we describe the three nodes in detail.

[1] The notation $W_{\Delta_i}^{\Delta_{i+c}}$ expresses a window of length $c$, which implies that the upper delimiter is excluded. This is done to simplify the notation as we otherwise would have to specify the upper limit as $i + c - 1$.

**Algorithm 2** Projection node $\pi$

**input:** stream of tuples $(x, y, \Delta_z)$ where $x, y$ are coordinates and $\Delta_z$ is the current period; precision $\xi$
**output:** stream of tuples $(x_\pi, y_\pi, \Delta_z)$
1: $x_\pi := 10^\xi x$
2: $y_\pi := 10^\xi y$
3: send $(x_\pi, y_\pi, \Delta_z)$ to node $\alpha$

#### *Projection*

Nodes labeled with $\boldsymbol{\pi}$ project incoming values to tiles with the specified precision (cf. Alg. 2). In general, the input $(x, y, \Delta_z)$ is transformed into $(x_\pi, y_\pi, \Delta_z)$. Projection of any value $v$ to $v_\pi$, using precision $\xi$, is defined as $v_\pi = 10^\xi v$, where $v, \xi \in \mathbb{R}, v_\pi \in \mathbb{N}$. This entails that $v_\pi$ is forcibly typecast to an integer. Thus, the fractional part of the scaled input is removed as this is the information we do not want to retain. The period $\Delta_z$ is a non-strictly increasing integer, for instance uniquely identifying each day. Projection is a stateless operation that can be executed in parallel. It is irrelevant which node $\pi$ performs projection on which input value as they are interchangeable. However, the input of the subsequent accumulator nodes $\boldsymbol{\alpha}$ needs to be grouped by values, for instance by assigning values within a certain segment of the longitude range of the input values. Yet, this is not strictly necessary as long as it is ensured that every unique surjected value $(x_\pi, y_\pi)$ is sent to the same $\alpha$-node.

#### *Accumulation*

The accumulator nodes $\boldsymbol{\alpha}$ keep track of the number of counts per projected tile in the sliding window $W_{\Delta_i}^{\Delta_{i+c}}$ (cf. Alg. 3). The input consists of a stream of tuples $(x_\pi, y_\pi, \Delta_z)$ as well as the size of the sliding window $c$ and the threshold value $\tau$. A global variable $\Delta_j$ keeps track of the current period. Each $\alpha$-node maintains two persistent data structures. For reasons of efficiency, the hashmap *totals* records, for each tile, how many points were projected to it in $W_{\Delta_i}^{\Delta_{i+c}}$. Tiles become significant once their associated count reaches $\tau$. In addition, the hashmap *window* records the counts per tile for each period $W_{\Delta_z}$ in $W_{\Delta_i}^{\Delta_{i+c}}$. Given that input $\Delta_z$ is non-strictly increasing, there are two cases to consider:

i. $\Delta_z = \Delta_j$, i.e. the period is unchanged. In this case, the count of tile $(x_\pi, y_\pi)$ in *totals* as well as its count corresponding to the key $\Delta_z$ in *window* are incremented by 1. If the total count for $(x_\pi, y_\pi)$ has just reached $\tau$, an update is sent to the $\kappa$-node, containing $(x_\pi, y_\pi)$ and the flag 1.
ii. $\Delta_z > \Delta_j$, i.e. the current input belongs to a later period. Now the sliding window needs to be advanced, which means that the oldest entry gets pruned. But first, an update is sent to the $\kappa$-node with $\Delta_j$ and the flag 0. Afterwards, the entries in the hashmap *window* have to be adjusted. Entry $\Delta_{z-c}$ is removed and for each coordinate pair and its associated counts, the corresponding entry in the hashmap *totals* gets adjusted downward as the totals should now no longer include these values. In case the associated value of a coordinate pair in *totals* drops below $\tau$, an update is sent to $\kappa$, consisting of $(x_\pi, y_\pi)$ and the flag -1. Should a value in *totals* reach 0,

the corresponding key-value pair is removed. Afterwards, the steps outlined in the previous case are performed.

Regarding significant tiles, only status changes are communicated from $\boldsymbol{\alpha}$ nodes to $\kappa$, which is much more efficient than sending a stream with constant status updates for each tile.

**Algorithm 3** Accumulation node $\alpha$

**input:** stream $s$ of tuples $(x_\pi, y_\pi, \Delta_z)$, sliding window size $c$, threshold $\tau$
**output:** stream of tuples $(flag, val)$ where $flag \in \{-1, 0, 1\}$ and $val \in \{(x_\pi, y_\pi), \Delta_j\}$

1: $totals := \varnothing$ ▷ $\{(x_\pi, y_\pi) : count\}$
2: $window := \varnothing$ ▷ $\{\Delta_j : \{(x_\pi, y_\pi) : count\}\}$
3: $\Delta_j := -1$ ▷ Period count
4: **for** $(x_\pi, y_\pi, \Delta_z)$ in $s$ **do** ▷ $\Delta_z - \Delta_j \in \{0, 1\}$
5: **if** $\Delta_z > \Delta_j$ **then** ▷ New period
6: send $(0, \Delta_j)$ to node $\kappa$ ▷ Re-cluster
7: $\Delta_j := \Delta_z$
8: $key := \Delta_j - c$ ▷ Oldest Entry
9: **if** $key \in$ keys of $window$ **then** ▷ Prune
10: $vals := window[key]$
11: **for** $(a, b)$ in keys of $vals$ **do**
12: $old := totals[(a, b)]$
13: $totals[(a, b)] \mathrel{-}= vals[(a, b)]$
14: $new := totals[(a, b)]$
15: **if** $old \geq \tau$ and $new < \tau$ **then**
16: send $(-1, (a, b))$ to node $\kappa$ ▷ Remove
17: **if** $new = 0$ **then**
18: remove entry $(a, b)$ from $totals$
19: remove entry $key$ from $window$
20: **if** $(x_\pi, y_\pi) \notin$ keys of $totals$ **then**
21: $totals[(x_\pi, y_\pi)] := 1$
22: $window[\Delta_z][(x_\pi, y_\pi)] := 1$
23: **else**
24: $totals[(x_\pi, y_\pi)] \mathrel{+}= 1$
25: **if** $(x_\pi, y_\pi) \notin$ keys of $window[\Delta_z]$ **then**
26: $window[\Delta_z][(x_\pi, y_\pi)] := 1$
27: **else**
28: $window[\Delta_z][(x_\pi, y_\pi)] \mathrel{+}= 1$
29: **if** $totals[(x_\pi, y_\pi)] = \tau$ **then**
30: send $(1, (x_\pi, y_\pi))$ to node $\kappa$ ▷ Add

### *Clustering*

The clustering node $\kappa$ (cf. Alg. 4) takes as input a precision value $\xi$, which is identical to the one that was used in the $\alpha$-node, the minimum cluster size $\mu$, and a stream consisting of tuples of a flag $\in \{-1, 0, 1\}$ as well as a value $val \in \{(x_\pi, y_\pi), \Delta_j\}$. This node keeps track of the significant tiles $\sigma$ of the current sliding window, based on updates received from all $\boldsymbol{\alpha}$ nodes. If $flag = 1$, the associated coordinate pair is added to $\sigma$. On the other hand, if $flag = -1$, tile $(x_\pi, y_\pi)$ is removed from $\sigma$. Thus, $\sigma$ is synchronized with the information stored in all $\boldsymbol{\alpha}$ nodes. Lastly, if $flag = 0$, the associated value of the input tuple represents a period identifier $\Delta_j$. This is interpreted as the beginning of this period and, conversely, the end of period $\Delta_{j-1}$. Now $\kappa$ clusters the set of significant tiles (cf. Alg. 1, lls. 12–24), taking $\mu$ into account, and produces an output stream that represents the clusters found within the current sliding window. In this stream, each projected coordinate $(x_\pi, y_\pi)$ is assigned period

and cluster identifiers. The coordinate pairs $(x_\pi, y_\pi)$ are re-scaled to floating point numbers $(x'_\pi, y'_\pi)$ by reversing the operation performed in node $\pi$ earlier. The output thus consists of a stream of tuples of the format $(\Delta_j, cluster\_id, x'_\pi, y'_\pi)$.

**Algorithm 4** Clustering node $\kappa$

**input:** stream $s$ of tuples $(flag, val)$ where $flag \in \{-1, 0, 1\}$ and $val \in \{(x_\pi, y_\pi), \Delta_j\}$, precision $\xi$, size $\mu$
**output:** stream of tuples $(\Delta_j, cluster\_id, x'_\pi, y'_\pi)$

```
1: σ := ∅                                              ▷ Significant tiles
2: for (flag, val) in s do
3:   if flag = 1 then
4:     (x_π, y_π) := val
5:     σ := σ ∪ {(x_π, y_π)}
6:   if flag = −1 then
7:     (x_π, y_π) := val
8:     σ := σ \ {(x_π, y_π)}
9:   if flag = 0 then                                  ▷ Next period
10:    Δ_j := val
11:    clusters := cluster(σ, μ)          ▷ cf. Alg. 1, lls. 12 – 24
12:    id := 0                                         ▷ Cluster ID
13:    for cluster in clusters do
14:      for (x_π, y_π) in cluster do
15:        (x'_π, y'_π) := rescale(x_π, y_π, ξ)        ▷ Int to Float
16:        send (Δ_j, id, x'_π, y'_π)
17:      id += 1
```

### Implementation details

The previous description is idealized. Yet, our software solution has to take the vagaries of real-world data into account. Below, we therefore highlight two relevant practical aspects that the formal definition of our algorithm does not capture.

#### *Out-of-order processing*

Tuples are assigned a timestamp at the source, which is projected to a period identifier $\Delta_z$. Assuming that the input stream is in-order, parallelizing the $\alpha$-operator could nonetheless lead to out-of-order input of some tuples at the $\kappa$ node, i.e. the latter could receive a notification about the start of period $\Delta_j$, cluster all significant tiles as they were recorded up to the seeming end of $\Delta_{j-1}$, but receive further tuples pertaining to it from other $\alpha$-nodes afterwards. One solution is to simply ignore these values as their number should be minuscule. Commonly used periods, e.g. one day, are quite large and the expected inconsistencies are confined to their beginning and end. Thus, it may make sense to set the start of a period to a time where very little data is generated, e.g. 3:00 a.m. for a 24-h period when processing data of commercial vehicles. Depending on actual use cases, other engineering solutions may be preferable. One promising approach would be to not immediately send a notification to initiate clustering to node $\kappa$ when the first element of a new period $\Delta_j$ is encountered by any $\alpha$ node. Instead, one could buffer a certain number of incoming elements that are tagged with $\Delta_j$ for a sufficiently large amount of time, call it a grace period. During that time, elements tagged with $\Delta_j$ are not processed yet. Instead, only elements that are tagged with $\Delta_{j-1}$ are. Once the grace period is up, a signal is sent to $\kappa$ to initiate clustering and the buffered elements in each $\alpha$ node are processed.

#### *Interpolating periods*

The algorithm as it is described does not take into account that there could be periods without new data. For instance, after processing the last data point in period $\Delta_x$, the next data point may belong to period $\Delta_{x+2}$. When pruning the sliding window, it is therefore not sufficient to only remove data for key $\Delta_{x+2-c}$, where $c$ is the size of the sliding window, as this would lead to data for $\Delta_{x+1-c}$ remaining in perpetuity. Thus, the sliding window also has to advance when there is no data for an entire period. If a gap greater than one period between the current and the last encountered period is detected, the algorithm has to advance the sliding window as many times as needed, one period at a time. After each period, the $\kappa$ node then clusters the significant tiles it has records of. Interpolation is omitted from Alg. 3 for the sake of brevity. Instead, we assume that there is at least one value associated with each element of the input stream. However, our published implementation is able to correctly prune the sliding window and update its clusters when it detects such a skip in the period counter.

#### Retaining data points with S-RASTER′

There are use cases where it is desirable to not only identify clusters based on their significant tiles but also on the data points that were projected to those tiles (cf. Fig. 1c, d). In the following, we refer to the variant of S-RASTER that retains relevant input data as S-RASTER′. The required changes are minor. With the goal of keeping the overall design of the algorithm unchanged, first the $\boldsymbol{\pi}$ nodes have to be modified to produce a stream of tuples $(x_\pi, y_\pi, x, y, \Delta_z)$, i.e. it retains the original input coordinates. In the $\boldsymbol{\alpha}$ nodes the hashmaps *totals* and *window* have to be changed to retain multisets of unscaled coordinates $(x, y)$ per projected pair $(x_\pi, y_\pi)$. Counts are given by the size of these multisets. This assumes that determining the size of a set is an $\mathcal{O}(1)$ operation in the implementation language, for instance due to the underlying object maintaining this value as a variable. In case a tile becomes significant, each $\alpha$-node sends not just the tile $(x_\pi, y_\pi)$ but also a bounded stream to $\kappa$ that includes all coordinate pairs $(x, y)$ that were projected to it up to that point in the current window. Furthermore, after a tile has become significant, every additional point $(x_\pi, y_\pi)$ that maps to it also has to be forwarded to $\kappa$, which continues for as long as the number of points surjected to that tile meet the threshold $\tau$. Lastly, in the $\kappa$ node, the set *tiles* has to be turned into a hashmap that maps projected tiles $(x_\pi, y_\pi)$ to their corresponding points $(x, y)$ and the output stream has to be modified to return, for each point $(x, y)$ that is part of a cluster, the tuple $(\Delta_j, cluster\_id, x_\pi, y_\pi, x, y)$.

## Theoretical evaluation

As we have shown previously, RASTER generates results more quickly than competing algorithms [33], with the caveat that the resulting clusters might not include elements at the periphery of clusters that other methods might include. Yet, the big benefit of our algorithm is its very fast throughput, being able to process an arbitrary amount of data in linear time and constant memory with a single pass. This makes it possible to use a standard workstation even for big data, while comparable workloads with other algorithms would necessitate a much more expensive and time-consuming cloud computing

setup; transferring terabytes of data to a data center alone is not a trivial matter, after all, even if we ignore issues of information sensitivity [17, 25]. The same advantages apply to S-RASTER as it is an adaptation of RASTER that does not change its fundamental properties. Our reasoning below shows that the performance benefits of RASTER for data batches, i.e. linear runtime and constant memory, carry over to S-RASTER for evolving data streams.

#### Linear runtime

RASTER is a single-pass linear time algorithm. The same is true for S-RASTER, which we will show by discussing the nodes $\pi$, $\alpha$, and $\kappa$ in turn. In general, data is continually processed as a stream, in which every input is only processed once by nodes $\pi$ and $\alpha$. These nodes perform a constant amount of work per incoming data point. The clustering node $\kappa$, however, performs two different actions. First, is continually updates the multiset of significant tiles $\sigma$ and, second, it periodically clusters $\sigma$. It is easy to see that there is at most one update operation per incoming data point, i.e. changing the multiplicity of the associated value of an incoming point in $\sigma$. In practice, the number of significant tiles $m$ is normally many orders of magnitude smaller than the number of input data points $n$. Furthermore, the threshold $\tau$ for a significant tile is normally greater than 1. Yet, even in the theoretically worst case it is not possible that $m > n$ as this would mean that at least one point was projected to more than one tile, which is not possible as the projection node $\pi$ performs a surjection. Consequently at worst there are $m = n$ significant tiles if there is exactly one point per significant tile and $\tau = 1$.

Clustering in $\kappa$ does not happen continually but periodically, i.e. in an offline manner. Furthermore, clustering significant tiles is a very fast procedure with a very modest effect on runtime. Even though Alg. 1, lls. 12–24, show a nested loop for clustering, this part is nonetheless linear because the inner loop removes elements from the input the outer loop traverses over. Thus, the entire input, which is the set of significant tiles $\sigma$, is traversed only once. In addition, lookup is a $\mathcal{O}(1)$ operation because the location of all neighboring tiles is known.

In the node $\kappa$, clustering is performed once per period, so take the number of input points per period $n$ and the number of significant tiles $m$. As we have shown, it is not possible that $m > n$, so at worst $m = n$. Clustering only happens once per period, for a total of $\mathcal{P}$ periods in the stream. Periods are not expressed in relation to $n$ but are dependent on time. Thus, $\mathcal{P}$ can be treated as a constant. The total runtime complexity of S-RASTER is $\mathcal{O}(n)$. Nodes $\pi$ and $\alpha$ perform a constant amount of work per data point. The same applies to the updating of multiplicities in $\kappa$, which is likewise a constant amount of work per data point. Lastly, the cost of clustering, which is an $\mathcal{O}(m)$ operation, is amortized over all data points in a period, which is a constant because clustering happens only periodically as opposed to continually, i.e. for each new data point. Expressed as a function on $n$, the time complexity of S-RASTER is, for $\pi$, $\alpha$, and the two parts of $\kappa$, $\mathcal{O}(n) + \mathcal{O}(n) + (\mathcal{O}(n) + \mathcal{O}(\mathcal{P}m)) = \mathcal{O}(n)$. On a related and more practical note, the performance impact of periodic clustering in the $\kappa$ node can be mitigated by running this node on a separate CPU core, which is straightforward in

the context of stream processing. This does not affect the previous complexity analysis, however.

#### Constant memory

RASTER needs constant memory $M$ to keep track of all counts per tile of the entire (finite) grid. In contrast, S-RASTER uses a sliding window of a fixed length $c$. In the worst case, the data that was stored in $M$ with RASTER requires $cM$ memory in S-RASTER, which is the case if there is, for each discrete period $\Delta_z$, at least one projected point per tile for each tile. Thus, S-RASTER maintains the key property of RASTER of using constant memory.

## Experiment

This section describes the design of our experiments, gives details of the experimental environment, and specifies how we performed a comparative empirical evaluation of S-RASTER and related clustering algorithms for data streams.

#### Design

For our experiments, we generated input files containing a fixed number of points arranged in dense clusters. These data sets contain no noise. The reason behind this decision is that RASTER is not affected by it. Noise relates to tiles that contain less than $\tau$ projected data points, which are simply ignored. In contrast, other algorithms may struggle with noisy data, in particular if they retain the entire input, which would potentially disadvantage them. Our primary goal was to measure clustering performance, i.e. speed. In addition, we took note of various standard clustering quality metrics such as the silhouette coefficient and distance measurements. Concretely, in the first experiment, the chosen algorithms process 5M data points. This set contains 1000 clusters. Every 500K points, we measure how long that part of the input data took to process. This implies a tumbling window, and for each batch of the input there are 100 different clusters, modeling an evolving data stream. This experiment is run ten times. In contrast, in the second experiment, we use a smaller input data set of 2K points. The variant of our algorithm for this experiment is SW-RASTER, which, in contrast to S-RASTER, does not have a notion of time but instead uses points to define window sizes. This modification was done because the algorithms we use for comparisons define the sliding window similarly. In this experiment, we use a number of established clustering quality metrics, i.e. the within-cluster sum of squares (SSQ) [14, p. 26], adjusted Rand index (cRand) [15], silhouette coefficient [28], and Manhattan distance.

#### Data set

S-RASTER was designed for handling a particular proprietary real-world data set, which we cannot share due to legal agreements. However, in order to evaluate this algorithm, we developed a data generator that can create an arbitrarily large synthetic data set that has similar properties. This data generator is available via our code repository. In short, the data generator randomly selects center points on a 2D plane and scatters points

around each such center point. There is a minimum distance between each center. For the experiment, we created a file with 1000 clusters of 500 points each, i.e. 500K points in total per batch. A batch corresponds to a period, e.g. one day. As we have processed ten batches, the total is 5M data points.

#### Environment

The used hardware was a workstation with an Intel Core i7-7700K and 32 GB RAM. Its main operating system is Microsoft Windows 10 (build 1903). However, the experiments were carried out with a hosted Ubuntu 16.04 LTS operating system that was executed in VirtualBox 5.2.8, which could access 24 GB RAM.

#### Comparative empirical evaluation

In order to compare S-RASTER to standard algorithms within its domain, we implemented the algorithm for use in Hahsler's popular R package `stream` [14]. For our purpose, the main benefit is that it contains an extensive suite for the evaluation of algorithms. Of the available algorithms in this R package, we selected DStream [7], DBstream [4, 13], and Windowed *k*-means Clustering, which was implemented by Hahsler himself. They were chosen because they are standard clustering algorithms for data streams.

DStream is a density-based clustering algorithm that uses an equally spaced grid. It estimates the density of each cell in a grid, which corresponds to a tile in our description of S-RASTER. For each cell, the density is computed based on the number of points per cells. Subsequently, they are classified as dense, transitional, or sporadic cells. A decaying factor ensures that cell densities reduce over time if no new points are encountered. DBStream uses a dissimilarity metric for data points. If a data point in the incoming data stream is below the given threshold value for dissimilarity of any of the hitherto identified micro-clusters, it is added to that cluster. Otherwise, this data point is the seed of a new cluster. Lastly, Windowed *k*-means clustering is an adaptation of the well-known *k*-means algorithm. It partitions the input based on a fixed number *k* of seeds.

For each algorithm, we chose parameters that delivered good clustering results, based on visual inspection. Concretely, this led to the following parameter values: DStream uses a grid size of 0.0003. DBStream uses a radius $r = 0.0002$, a minimum weight $Cm = 0.0$, and a gap time of 25,000 points. Windowed *k*-means uses a window length of 100 and $k = 100$. Any parameter we did not specify but is exposed via the `stream` package was used with its defaults. Lastly, for S-RASTER, we used a precision $\xi = 3.5$ and a window size $c = 10$.

## Results and discussion

In this section we present the results of our evaluation of S-RASTER as well as a discussion of these results.

#### Results

The results of the first experiment are shown in Fig. 4. We start with the most relevant results. As Fig. 4a shows, S-RASTER is faster than Windowed *k*-means, DBStream and DStream, processing each batch of 500K points in a little less than 1.5 s. In contrast, the

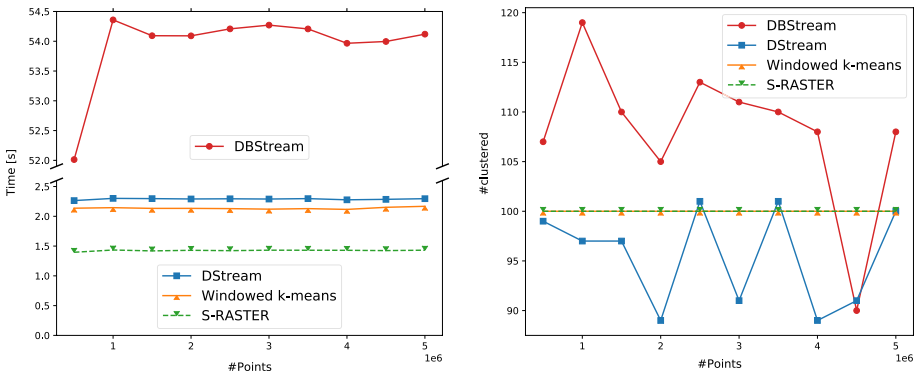
competing algorithm are at least 50% slower. DBStream takes about 40 times as long as S-RASTER. Figure 4b shows the number of clusters the various algorithms have found while processing the input data stream. The Windowed *k*-means algorithm was provided with an argument specifying $k = 100$. S-RASTER and Windowed *k*-means Clustering reliably identify 100 clusters per batch in the input data set whereas DBStream and DStream get close.

The results of the second experiment are summarized in Table 2, and visualized in Fig. 5. SW-RASTER, DBStream, and DStream deliver good results. Conceptually, the algorithms in the `stream` package identify clusters (macro clusters) that are based on smaller micro clusters, which may be defined differently, based on the chosen algorithm, e.g. squares in a grid or center points of a circle and their radius. Visual inspection seems to suggest that there are four macro clusters in the data set, which are made up of around 100 micro clusters. SW-RASTER delivers the densest and most separate clusters, which is expressed in the lowest adjusted Rand index (cRand) in this comparison. The cRand measure takes a value between 0 and 1. A value of 1 indicates complete similarity of two partitions, a value of 0 the opposite. SW-RASTER has a cRand value of 0.04, while the other clustering algorithms have cRand values of 0.06. In addition, together with Windowed *k*-means, SW-RASTER has the lowest silhouette coefficient with 0.18. Lastly, SW-RASTER has the lowest Manhattan distance of the chosen algorithms.

#### Discussion

We have shown that both in theory and practice the benefits of RASTER are retained for S-RASTER. S-RASTER is very fast, outperforming other clustering algorithms for data streams. Of course, the drawback is that there is some loss of precision. In other words, the comparatively good resulting metrics for various cluster quality measures are partly due to the algorithm ignoring points due to the chosen $\delta$ and $\sigma$ parameters. This is also reflected in the lower values for the SSQ and purity metrics, which are due to the algorithm ignoring points at the periphery. That being said, S-RASTER performs very well in the use case it has been designed for, which is not negatively affected by the trade-offs we made in the design of this algorithm. Also note that, at least theoretically, S-RASTER can be easily parallelized (cf. Sect. "Future work"). Yet, as the R package `stream` is a single-threaded library, this is not an angle we have pursued in this paper. After all, we would not have been able to reap any benefits from creating a multi-threaded implementation in this scenario.

It should be pointed out that both *k*-means clustering and S-RASTER consistently identify 100 clusters per batch in the input data stream. In the case of the former, this is due to the provided parameter $k = 100$, which invariably leads to the identification of 100 clusters. However, the case is much different with S-RASTER. The reason this algorithm identifies 100 clusters in each batch is that this algorithm was developed for reliably detecting dense clusters, ignoring noise and the sparser periphery of a collection of points that competing algorithms may classify as being part of the same cluster. Because there are 100 dense clusters in the input, S-RASTER was able to detect that number with suitable parameter values. In fact, it would have been cause of concern for us had this algorithm not reliably detected all clusters.




**(a)** Runtimes **(b)** Identified clusters

**Fig. 4** Evaluation of S-RASTER (best viewed in color). As **a** shows, S-RASTER is faster than competing algorithms in terms of throughput. The algorithms processed 5M points in 10 batches of 500K points. The stated times refer to the end of each batch. As a sanity check, **b** plots the number of identified clusters. The input data set contained 100 dense clusters per batch. Windowed *k*-means was provided with the argument $k = 100$. All data is the average of 10 runs

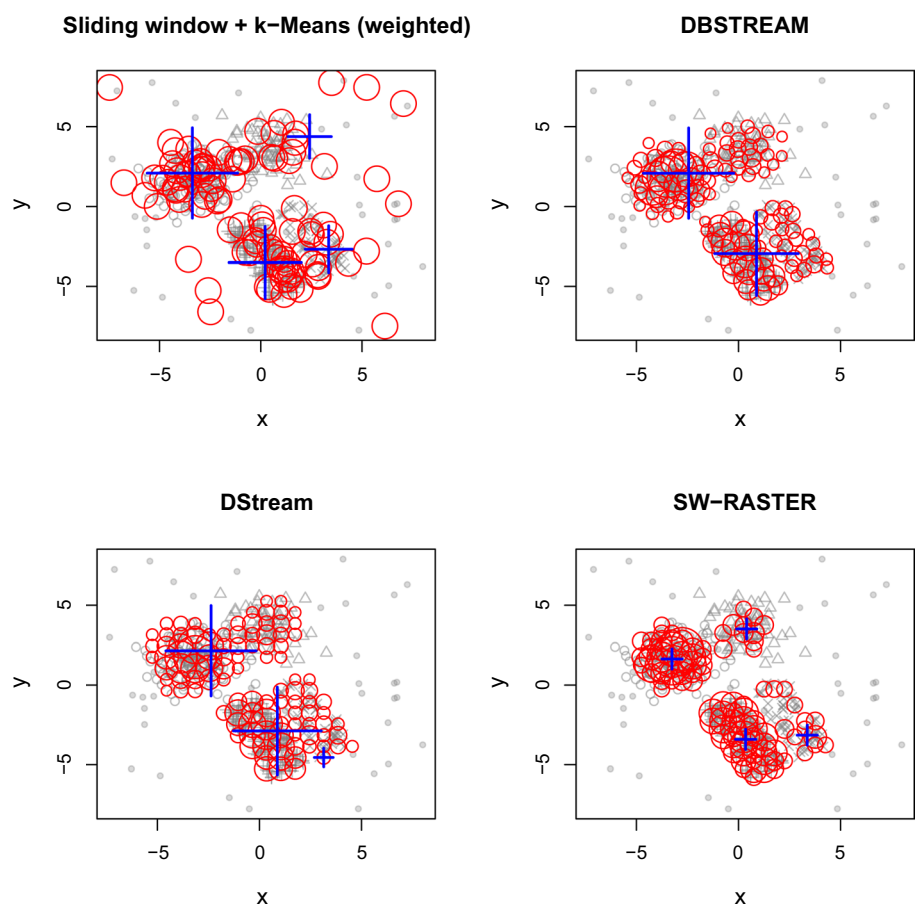



**Fig. 5** Illustration of clustering results with micro and macro clusters after clustering 500 points of the input data stream. S(W)-RASTER produces denser clusters than the other algorithms. The trade-off is that the algorithm ignores points at the periphery of the clusters, depending on the chosen parameters

**Table 2 Comparison of SW-RASTER with various standard clustering algorithms**

| | SW-RASTER | Windowed k-means | DStream | DBStream |
|---|---|---|---|---|
| Macro clusters | *4* | *4* | 3 | 2 |
| Micro clusters | 103 | 100 | 108 | 118 |
| purity | 0.93 | 0.94 | 0.96 | *0.97* |
| SSQ | 77.80 | 114.26 | 50.70 | *44.72* |
| cRand | *0.04* | 0.06 | 0.06 | 0.06 |
| silhouette | *0.18* | *0.18* | 0.21 | 0.27 |
| Manhattan | *00.11* | 0.12 | 0.13 | 0.13 |

The best values in this comparison are listed in italics. The number of micro clusters is listed for the sake of completion but we abstain from making a judgment as the resulting macro clusters are more relevant. Our algorithm does well in this comparison, as evinced by the cRand, silhouette coefficient and Manhattan distance values

## Related work

Two prominent related algorithms we did not consider in this paper are DUC-STREAM [11] and DD-Stream [16], which is due to the absence of a conveniently available open-source implementation. DUC-STREAM performs clustering based on dense unit detection. Its biggest drawback, compared to S-RASTER, is that it has not been designed for handling evolving data streams. While it also uses a grid, the computations performed are more computationally intensive than the ones S-RASTER performs, many of which are based on $\mathcal{O}(1)$ operations on hash tables. DD-Stream likewise performs density-based clustering in grids and likewise uses computationally expensive methods for clustering. DD-Stream is not suited for handling big data. Furthermore, unlike those algorithms, S-RASTER can be effectively parallelized (cf. Fig. 3). This primarily refers to the nodes $\pi$ and $\alpha$, which can be executed in an embarrassingly parallel manner.

None of the aforementioned clustering algorithms are quite comparable to S-RASTER, however, as they retain their input. Also, their clustering methods are generally more computationally costly. Thus, S-RASTER requires less time and memory. S-RASTER′ is closer to those algorithms as it retains relevant input data. As it does not retain all input, S-RASTER is only inefficient with regards to memory use in highly artificial scenarios. This is the case where there is at most one point projected to any square in the grid, which implies that the algorithm parameters were poorly chosen as the basic assumption is that many points are surjected to each significant tile. Ignoring pathological cases, it can thus be stated that S-RASTER is very memory efficient. Furthermore, clustering, in the $\kappa$ node, is a very fast operation. In summary, S-RASTER is, for the purpose of identifying density-based clusters in evolving data streams, more memory efficient than competing algorithms, and also less computationally intensive. Thus, it is a good choice if the trade-offs they make are acceptable for a given use case.

There is a superficial similarity between the output of self-organizing maps (SOMs) [19] and S-RASTER. We therefore want to clearly highlight the differences. First, SOMs belong to an entirely different category of algorithms, i.e. artificial neural networks (ANNs). They are likewise used for unsupervised learning, albeit there are adaptations for unsupervised online learning [10]. SOMs were initially developed for visualizing nonlinear relations in high dimensional data [20], but they have been applied to clustering problems [18, 35] and even suggested as a substitute for $k$-means clustering [3]. Practical clustering applications include, for instance, biomedical analyses [21] and water

treatment monitoring [12]. It may be that SOMs can achieve results similar to S-RASTER, but at an arguably much larger runtime and memory cost, given that distance matrices are the standard data structure and the fact that the so-called Best Matching Unit is determined by computing the minimum Euclidian distance between the input and the neuron weights. Thus, a standard SOM requires $\mathcal{O}(m)$ distance computations for each input point, where $m$ is the number of neurons in the ANN. In contrast, S-RASTER requires only a single projection operation for each input, plus the amortized cost of clustering at the end of each period. Lastly, it is not obvious how sliding windows would be represented with SOMs.

### Future work

This paper is accompanied by an implementation of S-RASTER for use in the R package `stream`. In the future, we may release an implementation of S-RASTER for use in Massive Online Analysis (MOA) [5]. In addition, we may release a complete stand-alone implementation of this algorithm that can be fully integrated into a standard stream processing engine such as Apache Flink [6] or Apache Spark [37, 38]. This is particularly relevant for an area we have not considered in this paper, i.e. the scalability of S-RASTER on many-core systems. As we have shown theoretically, S-RASTER is easily parallelizable. What is missing is to quantify the performance gains that can be expected. The reason we have not pursued this yet is that S-RASTER performs real-world workloads easily in sequential operation. Furthermore, we are interested in applying S-RASTER to data with higher dimensionality. As we elaborated elsewhere [33, Sect. 3.3], there are $2d$ lookups per dimension $d$. As we are primarily interested in processing 3D data, the additional total overhead due to lookup is modest. As S-RASTER was designed for solving a particular real-world problem, the theoretical objection of the curse of high dimensionality is not relevant.

As S-RASTER was developed in response to a concrete use case (cf. Sect. "Identifying evolving hubs"), we focussed on periodical clustering as this entailed only a negligible cost, in particular because clustering can be performed at convenient times (cf. Sect. "Implementation details"). However, other use cases may necessitate continual clustering. Thus, it seems worthwhile to explore modifications to the clustering node $\kappa$ that perform the clustering operations in a more efficient manner. We explored a few possibilities for parallelizing of the clustering algorithm in our work on batch processing with RASTER [33, Sect. 3.5], which would be a good starting point. Another promising idea would be to only selectively perform clustering. Right now, the entire set of significant tiles $\sigma$ is clustered at the end of a period. Yet, in real-world scenarios, it is quite likely that there are not many changes between periods, in particular if they are short. This is even more relevant when we consider the case of continual clustering. In those cases, a lot of redundant work would be performed if we clustered all of $\sigma$, considering that most clusters would not have changed much.

A potential application domain for RASTER and S-RASTER is image clustering, in particular with a focus on clustering multispectral images, which is a well-established area of research [31]. This would necessitate adding another dimension to

the algorithm, i.e. spectral as well as spatial information. There is a potentially wide domain of applications as multivariate images are very common in some domains. Two very prominent examples are magnetic resonance images and remote sensing images. *k*-means clustering has been successfully applied to this problem domain [24, 26]. One issue of *k*-means clustering, however, is its susceptibility to get trapped in local optima [22], which is not an issue for S-RASTER. It is also the case that *k*-means clustering is slower than S-RASTER. A particularly fruitful field of application for S-RASTER could be image-change detection, which has seen some interest [39], as some imprecision can be tolerated as long as changes are reliably detected.

## Conclusions

The key takeaway of our original work on RASTER was that by carefully chosen trade-offs, we are able to process geospatial big data on a local workstation. Depending on the use cases, those trade-offs may furthermore have a negligible impact on the precision of the results. In fact, in the case of the problem of identifying hubs in a batch of geospatial data, the loss of precision is immaterial. However, because RASTER is limited to processing batch data, we redesigned this algorithm as S-RASTER, using a sliding window. Thus, S-RASTER can be used to determine clusters within a given interval of the data in real-time. This algorithm is particularly relevant from an engineering perspective as we retain the same compelling benefits of RASTER, i.e. the ability to process data in-house, which leads to significant savings of time and cost compared to processing data at a remote data center. It also allows us to sidestep problems related to data privacy as business-critical geospatial data can now remain on-site. The trade-offs of S-RASTER compared to other streaming algorithms are also worth pointing out, as we, again, carefully designed its features with an eye to real-world applications. While many clustering algorithms for data streams continually update the clusters they identified, S-RASTER avoids this overhead by doing so only in fixed intervals, which is made possible by the very fast clustering method of RASTER, entailing an insignificant amortized cost. The overall result is that S-RASTER is very fast and delivers good results. Consequently, this algorithm is highly relevant for real-world big data clustering use cases.

**Acknowledgements**
This research project was carried out in the Fraunhofer Cluster of Excellence *Cognitive Internet Technologies*.

**Authors' contributions**
GU conceived the S-RASTER algorithm and its initial mathematical formulation, created a prototypal implementation of the algorithm in Kotlin, and wrote the manuscript. SS ported the existing implementation of S-RASTER to C++, and conducted experiments. He was assisted by AN who was also involved in the literature review. All authors were involved in the experimental design and interpretation of results. MJ contributed to the mathematical formulations in this paper. All authors read and approved the final manuscript.

**Funding**
This research was supported by the project ***Fleet telematics big data analytics for vehicle usage modeling and analysis*** (FUMA) in the funding program ***FFI: Strategic Vehicle Research and Innovation*** (DNR 2016-02207), which is administered by VINNOVA, the Swedish Government Agency for Innovation Systems.

**Data availability**
The datasets generated and analyzed during the current study are available in the following source code repository: https://github.com/FraunhoferChalmersCentre/s-raster. This repository contains a data generator and the entire code used for benchmarking the algorithm, including a complete implementation of S-RASTER. Additionally, we provide a reference implementation of RASTER in Python as well as a proof-of-concept implementation of S-RASTER in Kotlin.

**Competing interests**
The authors declare that they have no competing interests.

**Author details**
[1] Fraunhofer-Chalmers Research Centre for Industrial Mathematics, Chalmers Science Park, 412 88 Gothenburg, Sweden.
[2] Fraunhofer Center for Machine Learning, Chalmers Science Park, 412 88 Gothenburg, Sweden.

Received: 26 April 2020 Accepted: 30 July 2020
Published online: 13 August 2020

**References**

1. Agrawal R, Gehrke J, Gunopulos D, Raghavan P. Automatic subspace clustering of high dimensional data for data mining applications. In: Proceedings of the 1998 ACM SIGMOD international conference on management of data. SIGMOD '98. New York: ACM; 1998. p. 94–105.
2. Agrawal R, Gehrke J, Gunopulos D, Raghavan P. Automatic subspace clustering of high dimensional data. Data Min Knowl Discov. 2005;11(1):5–33.
3. Bação F, Lobo V, Painho M. Self-organizing maps as substitutes for k-means clustering. In: International conference on computational science. Berlin: Springer; 2005. p. 476–83.
4. Bär A, Finamore A, Casas P, Golab L, Mellia M. Large-scale network traffic monitoring with dbstream, a system for rolling big data analysis. In: 2014 IEEE international conference on big data (big data). New York: IEEE; 2014. p. 165–70.
5. Bifet A, Holmes G, Kirkby R, Pfahringer B. Moa: massive online analysis. J Mach Learn Res. 2010;11(May):1601–4.
6. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache flink: stream and batch processing in a single engine. Bull IEEE Comput Soc Tech Comm Data Eng. 2015;36(4):28–38.
7. Chen Y, Tu L. Density-based clustering for real-time stream data. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining. New York: ACM; 2007. p. 133–42.
8. van Diggelen F, Enge P. The world's first gps mooc and worldwide laboratory using smartphones. In: Proceedings of the 28th international technical meeting of the satellite division of the institute of navigation (ION GNSS+ 2015). ION 2015. p. 361–9.
9. Ester M, Kriegel HP, Sander J, Xu X, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. SIGKDD Conf Knowl Discov Data Min. 1996;96:226–31.
10. Furao S, Ogura T, Hasegawa O. An enhanced self-organizing incremental neural network for online unsupervised learning. Neural Netw. 2007;20(8):893–903.
11. Gao J, Li J, Zhang Z, Tan PN. An incremental data stream clustering algorithm based on dense units detection. In: Pacific-Asia conference on knowledge discovery and data mining. Berlin: Springer; 2005. p. 420–5.
12. García HL, González IM. Self-organizing map and clustering for wastewater treatment monitoring. Eng Appl Artif Intell 17(3):215–225
13. Hahsler M, Bolaños M. Clustering data streams based on shared density between micro-clusters. IEEE Trans Knowl Data Eng. 2016;28(6):1449–61.
14. Hahsler M, Bolanos M, Forrest J, et al. Introduction to stream: An extensible framework for data stream clustering research with r. J Stat Softw. 2017;76(14):1–50.
15. Hubert L, Arabie P. Comparing partitions. J Classif. 1985;2(1):193–218.
16. Jia C, Tan C, Yong A. A grid and density-based clustering algorithm for processing data stream. In: 2008 second international conference on genetic and evolutionary computing. New York: IEEE; 2008. p. 517–21.
17. Kaisler S, Armour F, Espinosa JA, Money W. Big data: Issues and challenges moving forward. In: 2013 46th Hawaii international conference on system sciences. 2013. p. 995–1004.
18. Kiang MY. Extending the kohonen self-organizing map networks for clustering analysis. Comput Stat Data Anal. 2001;38(2):161–80.
19. Kohonen T. Self-organized formation of topologically correct feature maps. Biol Cybern. 1982;43(1):59–69.
20. Kohonen T. Applications. In: Self-organizing maps. Berlin: Springer; 2001. p. 263–310.
21. Kohonen T. Essentials of the self-organizing map. Neural Netw. 2013;37:52–65 twenty-fifth Anniversay Commemorative Issue.
22. Li H, He H, Wen Y. Dynamic particle swarm optimization and k-means clustering algorithm for image segmentation. Optik. 2015;126(24):4817–22.
23. MacQueen J et al. Some methods for classification and analysis of multivariate observations. In: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, Oakland, CA, USA, vol. 1. 1967. p. 281–97.
24. Mat Isa NA, Salamah SA, Ngah UK. Adaptive fuzzy moving k-means clustering algorithm for image segmentation. IEEE Trans Consumer Electron. 2009;55(4):2145–53.
25. Mazumdar S, Seybold D, Kritikos K, Verginadis Y. A survey on data storage and placement methodologies for cloud-big data ecosystem. J Big Data. 2019;6(1):15.
26. Ng HP, Ong SH, Foong KWC, Goh PS, Nowinski WL. Medical image segmentation using k-means clustering and improved watershed algorithm. In: 2006 IEEE southwest symposium on image analysis and interpretation. 2006. p. 61–5.
27. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, et al. Scikit-learn: machine learning in python. J Mach Learn Res. 2011;12(Oct):2825–30.
28. Rousseeuw PJ. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J Comput Appl Math. 1987;20:53–65.
29. Xu Rui, Wunsch D. Survey of clustering algorithms. IEEE Trans Neural Netw. 2005;16(3):645–78. https://doi.org/10.1109/TNN.2005.845141.
30. Taylor G, Brunsdon C, Li J, Olden A, Steup D, Winter M. Gps accuracy estimation using map matching techniques: applied to vehicle positioning and odometer calibration. Comput Environ Urban Syst. 2006;30(6):757–72. https://doi.org/10.1016/j.compenvurbsys.2006.02.006.
31. Tran TN, Wehrens R, Buydens LM. Clustering multispectral images: a tutorial. Chemom Intell Lab Syst. 2005;77(1–2):3–17.

32. Ulm G, Gustavsson E, Jirstrand M. Contraction clustering (RASTER). In: Nicosia G, Pardalos P, Giuffrida G, Umeton R, editors. Machine learning, optimization, and big data. Cham: Springer International Publishing; 2018. p. 63–75.
33. Ulm G, Smith S, Nilsson A, Gustavsson E, Jirstrand M. Contraction clustering (RASTER): a very fast big data algorithm for sequential and parallel density-based clustering in linear time, constant memory, and a single pass. 2019. arXiv preprint arXiv:1907.03620.
34. Venkatasubramanian S. Clustering on streams. New York: Springer; 2018. p. 488–94.
35. Vesanto J, Alhoniemi E. Clustering of the self-organizing map. IEEE Trans Neural Netw. 2000;11(3):586–600.
36. Wing MG, Eklund A, Kellogg LD. Consumer-grade global positioning system (GPS) accuracy and reliability. J. For. 2005;103(4):169–73. https://doi.org/10.1093/jof/103.4.169.
37. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. HotCloud. 2010;10(10–10):95.
38. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, et al. Apache spark: a unified engine for big data processing. Commun ACM. 2016;59(11):56–65.
39. Zheng Y, Zhang X, Hou B, Liu G. Using combined difference image and *k*-means clustering for sar image change detection. IEEE Geosci Remote Sens Lett. 2013;11(3):691–5.


## Publisher's Note